

Chapter 7: Classes (A First Look)

(ppt)

Overview

In C++ we use **classes** to define **our own data types**. By defining types that mirror concepts in the problems we are trying to solve, we can make our programs easier to write, debug, and modify.

This chapter continues the coverage of classes begun in Chapter 2 (`struct`). Here we will focus on the importance of **data abstraction**, which lets us **separate** the implementation of an object from the operations that that object can perform. In Chapter 13 we'll learn how to control what happens when objects are copied, moved, assigned, or destroyed. In Chapter 14 we'll learn how to define our own operators.

Fundamental Ideas Behind Classes and Abstract Data Types (ADT)

The fundamental ideas behind classes are data abstraction and encapsulation. **Data abstraction** means representation of information in terms of its interfaces with the user (e.g., `push_back` in `vector`). **Encapsulation** means hiding details of implementation (e.g., internal memory management in `vector`).

A class that uses data abstraction and encapsulation is an **abstract data type**. In an abstract data type, the class designer worries about how the class is implemented. Programmers who use the class need not know how the type works. **They can instead think abstractly about what the type does.**

Q: Does the `Student` data type listed below an **abstract** data type?

```
struct Student
{
    std::string name;
    int id;
    int age;
};
```

A:

(see ppt on Different Kinds of Programming Role in C++)

7.1 Defining Abstract Data Types

7.1.1 Designing the Sales_data Class

Recall what we have on Sales_data class (2.6.1, pp. 72):

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

We will add operations for data abstraction. The interface to Sales_data consists of the following operations:

- An isbn **member function** to return the object's ISBN
- A combine **member function** to add one Sales_data object into another
- An add function to add two Sales_data objects
- A read function to read data from an istream into a Sales_data object
- A print function to print the value of a Sales_data object on an ostream.

And the users might use these interfaces like the following:

```
Sales_data total; // variable to hold the running sum
if (read(cin, total)){ // read the first transaction
    Sales_data trans; // variable to hold data for the next transaction
    while(read(cin, trans)){ // read the remaining transactions
        if (total.isbn() == trans.isbn()) // check the isbn
            total.combine(trans); // update the running total
        else {
            print(cout, total) << endl; // print the results
            total = trans; // process the next book
        }
    }
    print(cout, total) << endl; // print the last transaction
} else { // there was no input
    cerr << "No data?!" << endl; // notify the user
}
```

Once we learn how to define our own operators (Ch. 14), we can use Sales_item as we have done in Ch. 1.

```

Sales_item total; // variable to hold the running sum
if (cin >> total) { // read the first transaction
    Sales_item trans; // variable to hold data for the next transaction
    while (cin >> trans) { // read the remaining transactions
        if (total.isbn() == trans.isbn()) // check the isbn
            total += trans; // update the running total
        else {
            cout << total << endl; // print the results
            total = trans; // process the next book
        }
    }
    cout << total << endl; // print the last transaction
} else {
    cerr << "No data?!" << endl; // notify the user
}

```

7.1.2 Defining the Revised Sales_data Class and Member Functions

As we have seen from the **use** of the abstract data type, the revised `Sales_data` class will have two **member functions**, `combine` and `isbn`. In addition, we'll give `Sales_data` another member function to return the average price at which the books were sold. This function, which we'll name `avg_price`, isn't intended for general use. It will be part of the implementation, not part of the interface.

We define and declare **member functions** similarly to ordinary functions. Member functions **must be declared** inside the class. Member functions may be **defined** inside the class itself or outside the class body. **Nonmember functions** that are part of the interface, such as `add`, `read`, and `print`, are **declared and defined** outside the class.

With the above knowledge, the revised `Sales_data` class is now ready:

```

struct Sales_data {
    // new members: operations on Sales_data objects
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    // data members are unchanged from § 2.6.1 (p. 72)
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream& print(std::ostream&, const Sales_data&);

```

```
| std::istream& read(std::istream&, Sales_data&);
```

Defining Member Functions

Member functions may be defined inside the class itself or outside the class body. In general, if the function body consists of more than three lines, we should do it outside the class.

(1) Declare and define member function inside the class definition

```
| struct Sales_data {
|     std::string isbn() const { return bookNo; }
|     ...
| }
```

(2) Declare member function inside the class definition and define it outside

```
| struct Sales_data {
|     ...
|     double avg_price() const;
|
| double Sales_data::avg_price() const {
|     if (units_sold)
|         return revenue/units_sold;
|     else
|         return 0;
| }
```

(Important) Notice that the member functions defined outside the class definition look like ordinary functions except that they contain the class name and a double colon (::) before the function name (but after the return type). The :: symbol is called the **scope resolution operator**. It is needed to indicate that these are class member functions and to tell the compiler which class they belong to.

Ex 7.1: In-class Coding Exercise


Write a class named `Person` that represents the name and age of a person. Use a string and an unsigned to hold each of these elements. Provide member functions `setName` and `setAge` to set proper values of name and age. Provide member functions `getName` and `getAge` in your `Person` class to return the name and age. You should put the class definition in a header file `Person.h` with a proper header guard and `const` correctness. A sample client code and output look like:

PersonClient.cpp

```
| #include <iostream>
| #include "Person.h"
```

```
using namespace std;

int main()
{
    Person p;
    p.setName("John");
    p.setAge(25);
    cout << "Name: " << p.getName() << " age: " << p.getAge() << endl;
    return 0;
}
```

A screenshot of a terminal window with a black background and white text. The text displayed is "Name: John age: 25".

A:

Person.h

Knowing Your Objects: The `this` Pointer

The `combine` member function is intended to act like the compound assignment operator, `+=`. The object on which this member function is called represents the left-hand operand of the assignment. The right-hand operand is passed as an explicit argument:

```
| total += trans; // update the running total

| total.combine(trans); // update the running total

| Sales_data& Sales_data::combine(const Sales_data &rhs)
| {
|     units_sold += rhs.units_sold; // add the members of rhs into
|     revenue += rhs.revenue; // the members of 'this' object
|     return ???; // return the object on which the function was called
| }
```

We will need to **return the object on which the function was called**. The C++ solution to this problem is to use a special pointer called `this`. The `this` pointer points to the object used to invoke a member function. (Basically, `this` is passed as a hidden argument to the method.)

Thus, the function call `total.combine(trans)` sets `this` to the address of the `total` object and makes that pointer available to the `combine` method. In general, all class methods have a `this` pointer set to the address of the object that invokes the method. Indeed, `units_sold` in the `combine` member function is just shorthand notation for `this->units_sold`.

Q: Now complete `???` in the definition of `combine` member function.

A:

Ex 7.2: Quick Check on Concept

Consider the previous exercise on the class `Person`. If we would like to concatenate a sequence of `set` actions into a single expression so we can do:

```
| Person p;
| p.setName("John").setAge(25);
| p.setAge(25).setName("John");
```

Modify your `setAge` and `setName` implementation in the `Person` class to accomplish these.

```
void setName(const std::string& s){name = s;}  
void setAge(unsigned num){age = num;}
```

Answer:

const Member Functions

```
struct Sales_data {  
    // new members: operations on Sales_data objects  
    std::string isbn() const { return bookNo; }  
    Sales_data& combine(const Sales_data&);  
    double avg_price() const;  
    ...  
}
```

The `const` that follows the parameter listed in the declarations of the `Sales_data` member functions is called a const member function. `const` relates to this. this is a pointer to `const` so a `const` member function **cannot change the object on which it is called**. It implies that that `avg_price` and `isbn` may read but not modify/write to the data members of the objects on which they are called.

Q: Can the following code pass the compilation?

```
double Sales_data::avg_price() const {  
    this->units_sold = 10.0;  
    if (units_sold)  
        return revenue/units_sold;  
    else  
        return 0;  
}
```

A:

7.1.3 Defining Non-member Class-Related Functions

Class authors often define **auxiliary functions**, such as the non-member `add`, `read`, and `print` functions. Although such functions define operations that are conceptually part of the interface of the class, they are not part of the class itself.

We define **nonmember functions** as we would do for any other function. As with any other function, we normally separate the declaration of the function from its definition (§6.1.2, p. 206). Functions that are conceptually part of a class, but not defined inside the class, are typically declared (but not defined) in **the same header** as the class itself. That way users need to include only one file to use any part of the interface.

```

struct Sales_data {
    ...
};

// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);

ostream &print(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}

print(cout, total) << endl; // print the results

```

Notice that the IO classes are types that **cannot be copied**, so we may only pass them by reference. Moreover, reading or writing to a stream changes that stream, so both functions take ordinary references, not references to const.

7.1.4 Constructors

(see ppt)

A constructor is a special member function that is called whenever an object of the class is created. In C++, constructors are member functions with **the same name as the class**; they have **no return type**.

Synthesized Default Constructor

Classes control default initialization by defining a special constructor, known as the **default constructor**. The default constructor is one that takes no arguments.

The default constructor is special and if our class does not explicitly define any constructors, the compiler will implicitly define the default constructor for us.

The compiler-generated constructor is known as the **synthesized default constructor**. This synthesized constructor initializes each data member of the class as follows:

- If there is an in-class initializer, use it to initialize the member.
- Otherwise, default-initialize the member.

Quick Concept Check: How does the synthesized default constructor initialize the data members `a` and `b` in struct `A`?

```
| struct A {
|     int a;
|     std::string b;
| };
```

A:

For our `Sales_data` class we'll define **three** constructors with the following parameters:

- A `const string&` representing an ISBN, an `unsigned` representing the count of how many books were sold, and a `double` representing the price at which the books sold.
- A `const string&` representing an ISBN. This constructor will use default values for the other members.
- An empty parameter list (i.e., the default constructor) which as we've just mentioned; we must define the default constructor by ourselves because we have defined other constructors.

```
| struct Sales_data {
|     // constructors added
|     Sales_data() = default;
|     Sales_data(const std::string &s): bookNo(s) { }
|     Sales_data(const std::string &s, unsigned n, double p):
|         bookNo(s), units_sold(n), revenue(p*n) { }
|     ...
| }
```

What = default Means

```
| Sales_data() = default;
```

Under the new standard (C++11), if we simply want the default behavior, we can ask the compiler to generate the constructor for us by writing `= default` after the parameter list.

Constructor Initializer List

```

Sales_data(const std::string &s): bookNo(s) { }
Sales_data(const std::string &s, unsigned n, double p):
    bookNo(s), units_sold(n), revenue(p*n) { }

```

The colon up to the open curly is the constructor initializer list. A constructor initializer list specifies initial values for one or more data members of the object being created. The constructor initializer is a list of member names, each of which is followed by that member's initial value in parentheses (or inside curly braces). Multiple member initializations are separated by commas.

Quick Concept Check: consider a header file `democlass.h` and a class definition `DemoClass`

democlass.h

```

#ifndef DEMOCLASS_H
#define DEMOCLASS_H
struct DemoClass
{
    // add a constructor
    ... DemoClass(int a, int b): itemA(a), itemB(b) {}
    int itemA, itemB;
};
#endif

```

Add a default constructor using an **initializer list**. We shall (1) initialize `itemA` with the value of 0 and `itemB` with the value of 1 and (2) in the initializer list, initialize `itemA` with the first parameter and `itemB` with the second parameter.

A: **`DemoClass(int a=0, int b=1): itemA(a), itemB(b) {}`**
default constructor using an initialize list

Ex 7.3: In-class Coding Exercise
Lab

CircleClient.cpp

Write a simple `Circle` class (`Circle.h` and `Circle.cpp`) so one can set its radius through a constructor or by a member function. In addition, the `Circle` object can report its radius and area when we print the object. Below are a sample client code and output:

```

#include <iostream>
#include "Circle.h"
using namespace std;

```

```
int main()
{
    Circle c1;
    print (cout, c1);
    c1.setRadius(1);    // This sets c1 radius to 1
    print (cout, c1);
    Circle c2(2.5);    // This sets c2 radius to 2.5
    print (cout, c2);
    return 0;
}
```

```
Circle radius: 0 area: 0
Circle radius: 1 area: 3.14159
Circle radius: 2.5 area: 19.6349
```

Answer:

7.2 Access Control and Encapsulation

(see ppt)

In C++ we use **access labels** to enforce encapsulation. A class may contain zero or more access labels:

- Members defined after a **public** label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined after a **private** label are accessible by the member functions of the class, but are not accessible to codes that use the class. The private sections encapsulate (e.g., hide) the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

A class may define members before any access label is seen. The access level of members defined after the open curly of the class and before the first access label depends on how the class is defined. If the class is defined with the `struct` keyword, then members defined before the first access label are public; if the class is defined using the `class` keyword, then the members are private.

Quick Concept Check: which members are private members in the following?

```
class Fraction {  
    private:  
        int numer;  
        int denom;  
    public:  
        Fraction(int);  
        void print();  
    private:  
        Fraction();  
};
```

```
struct Fraction {  
    int numer;  
    int denom;  
};
```

```
class Fraction {  
    int numer;  
    int denom;  
};
```

We can now redefine `Sales_data` again with proper access control for encapsulation.

```
class Sales_data {
    public: // access specifier added
        Sales_data() = default;
        Sales_data(const std::string &s, unsigned n, double p):
            bookNo(s), units_sold(n), revenue(p*n) { }
        Sales_data(const std::string &s): bookNo(s) { }
        Sales_data(std::istream&);
        std::string isbn() const { return bookNo; }
        Sales_data &combine(const Sales_data&);
    private: // access specifier added
        double avg_price() const
            { return units_sold ? revenue/units_sold : 0; }
        std::string bookNo;
        unsigned units_sold = 0;
        double revenue = 0.0;
};
```

Quick Guideline: In designing a class, we typically put **data** members into the private section and put **member functions** into the public section. A typical class definition has the following form:

```
class className
{
    private:
        data member declarations
    public:
        member function prototypes
};
```

7.2.1 Friends

Now that the data members of `Sales_data` are private, our `read`, `print`, and `add` non-member functions will no longer compile. Why?

A:

A class can allow another class or function to access its nonpublic members by making that class or function a **friend**. A class makes a function its friend by including a declaration for that function preceded by the keyword **friend**:

```
class Sales_data {
    // friend declarations for nonmember Sales_data operations added
    friend Sales_data add(const Sales_data&, const Sales_data&);
    friend std::istream &read(std::istream&, Sales_data&);
    friend std::ostream &print(std::ostream&, const Sales_data&);
};
```

```

// other members and access specifiers as before
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
// declarations for nonmember parts of the Sales_data interface
Sales_data add(const Sales_data&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);

```

Quick Exercise: let us modify Ex7.3 Circle class to incorporate private and public accessible concepts:

```

#include <iostream>
#include "Circle.h"
using namespace std;

int main()
{
    Circle c1;
    print (cout, c1);
    c1.setRadius(1);    // This sets c1 radius to 1
    print (cout, c1);
    Circle c2(2.5);    // This sets c2 radius to 2.5
    print (cout, c2);
    return 0;
}

```

```

Circle radius: 0 area: 0
Circle radius: 1 area: 3.14159
Circle radius: 2.5 area: 19.6349

```


Q: Will the code be compilable?

A:

You can fix the problem by (1) adding a simple public utility function `getRadius` (2) or to declare `print` a friend of class `Circle`.

<Solution 1>

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle
{
    public:
        Circle() = default;
        Circle(double r){radius = r;}
        void setRadius(double r){radius = r;}
        double getArea() const {return PI*radius*radius;}
        double getRadius() const {return radius;}
    private:
        double radius = 0.0;
        const double PI = 3.14159;
};

std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c)
{
    os << "Circle radius: " << c.getRadius() << " area: "
      << c.getArea() << endl;
    return os;
}
```

<Solution 2>

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle
{
```



```
public:
    friend std::ostream &print(std::ostream &os, const Circle &c);
    Circle() = default;
    Circle(double r){radius = r;}
    void setRadius(double r){radius = r;}
    double getArea() const {return PI*radius*radius;}
private:
    double radius = 0.0;
    const double PI = 3.14159;
};

std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c)
{
    os << "Circle radius: " << c.radius << " area: "
        << c.getArea() << endl;
    return os;
}
```